Introduction to OpenCL

Framework, language, and examples

Session 307

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

Introduction to OpenCL

Part 1: Compute model and frameworks

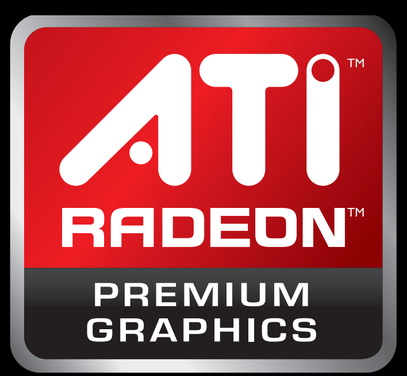
Part 2: OpenCL C kernel language

OpenCL Compute Model and Framework

David Black-Schaffer, Ph.D.
OpenCL Engineer

Welcome to OpenCL

- With OpenCL you can…
  - Leverage CPUs and GPUs to accelerate parallel computation
  - Get dramatic speedups for computationally intensive applications
  - Write accelerated portable code across different devices and architectures



What You'll Learn

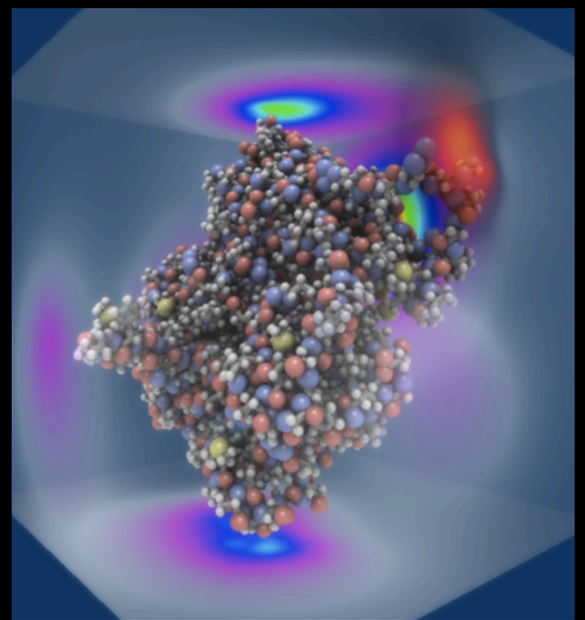
- What is OpenCL and what is it good for?
  - What types of algorithms map well to OpenCL
- How does OpenCL expose hardware?
  - The OpenCL compute and memory models
- How to use the OpenCL Framework
  - Setup
  - Resource allocation
  - Execution
  - Synchronization

What Is OpenCL?

What Is OpenCL?

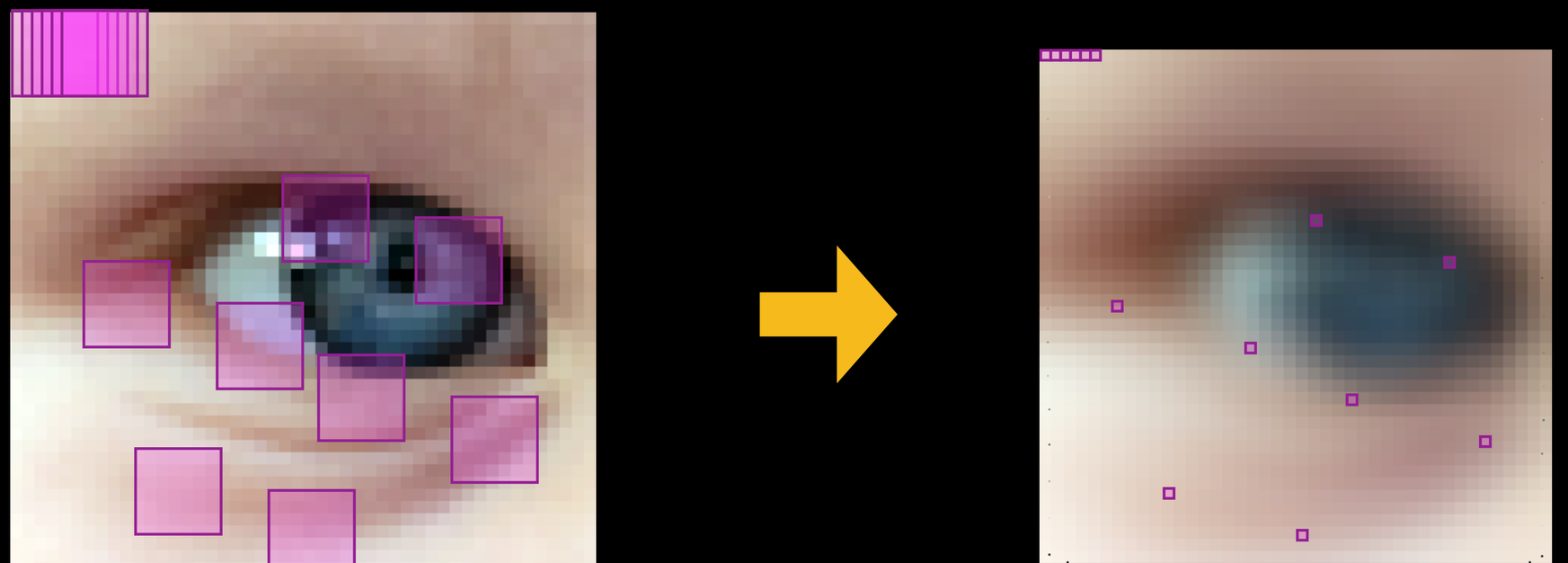
- Access to all compute resources in the system
- Low-level framework for high performance
- Targets (data)-parallel, computationally intensive algorithms

- Examples
  - Physics simulation
  - Image processing
  - Signal processing
  - Video and audio encoding
  - Math libraries
  - Ray tracing
  - Medical imaging
  - Financial modeling



What Is Data-Parallel Computing?

- Similar and independent computations across all data
- Example: Averaging an image (box filter)
  - Same computation for each output
  - All results are independent



Data-Parallel Kernel: 7 x 7 Average

```
kernel average(read_only image2d_t input,
               write_only image2d_t output)
{
   sampler_t sampler = CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;
   int x = get_global_id(0);
   int y = get_global_id(1);

   float4 sum = (float4)0.0f;
   int2 pixel;
   for (pixel.x=x-3; pixel.x<=x+3; pixel.x++)
    for (pixel.y=y-3; pixel.y<=y+3; pixel.y++)
     sum += read_imagef(input, sampler, pixel);

   write_imagef(output, (int2)(x, y), sum/49.0f);
}
```

Determine the pixel to work on

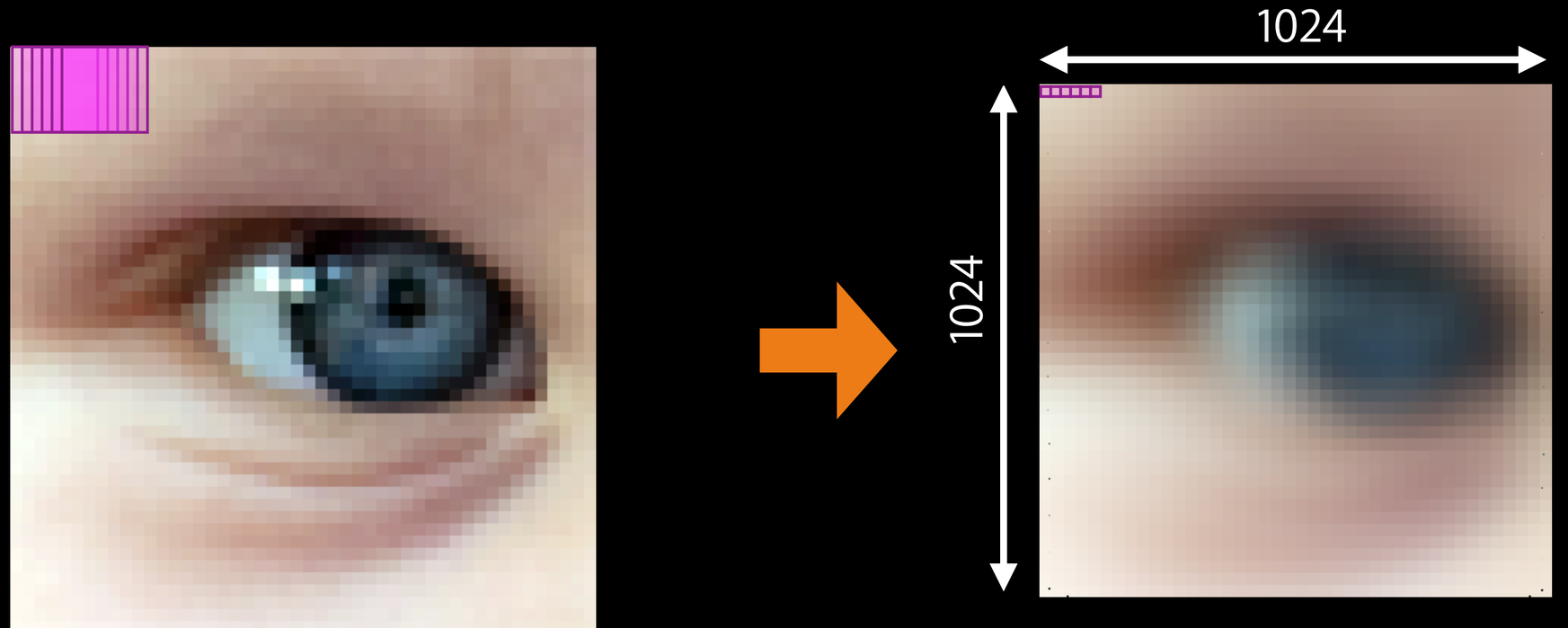
Process the region

Write out the average

Kernel executed once for every pixel by OpenCL

Expressing Parallelism in OpenCL

- Data-parallel across two dimensions
- Compute kernel executes once for every element
- E.g., process a 1024 x 1024 image:
  Global problem dimensions: 1024 x 1024 =
  1 kernel execution per pixel: 1,048,576 total kernel executions



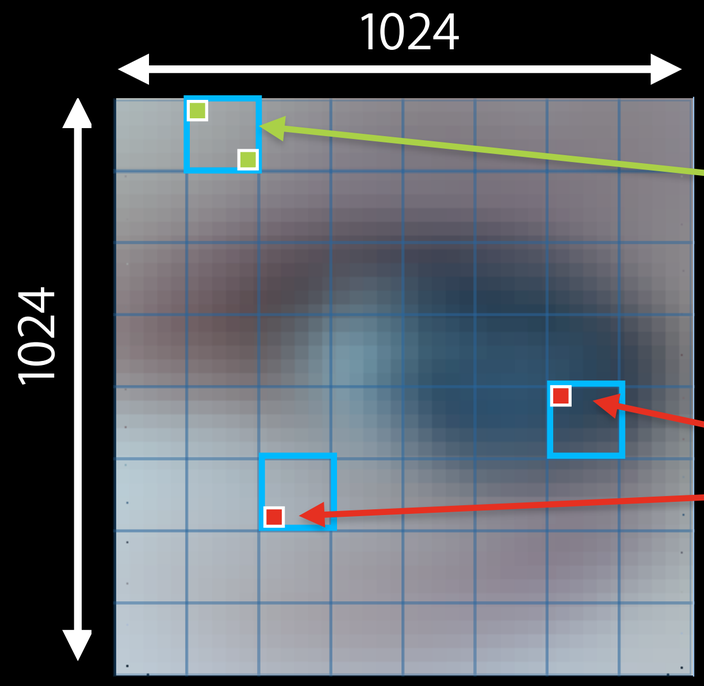
OpenCL Computation Model

- Kernels executed across a global domain of *work-items*
  - Global dimensions define the range of computation (e.g., 1024 x 1024)
  - One *work-item* per computation, executed in parallel (e.g., 1,048,576)
- Work-items are grouped in local *workgroups*
  - Local dimensions define the size of the *workgroups*
  - Executed together on one processor
  - Share local memory and synchronization
- Caveats
  - Global work-items must be independent: No global synchronization
  - **Synchronization can be done within a workgroup**

Global and Local Dimensions

Global Dimensions: 1024 x 1024 (whole problem space)

Local Dimensions: 128 x 128 (executed together)



Synchronization between *work-items* possible only **within** *workgroups*: barriers and memory fences

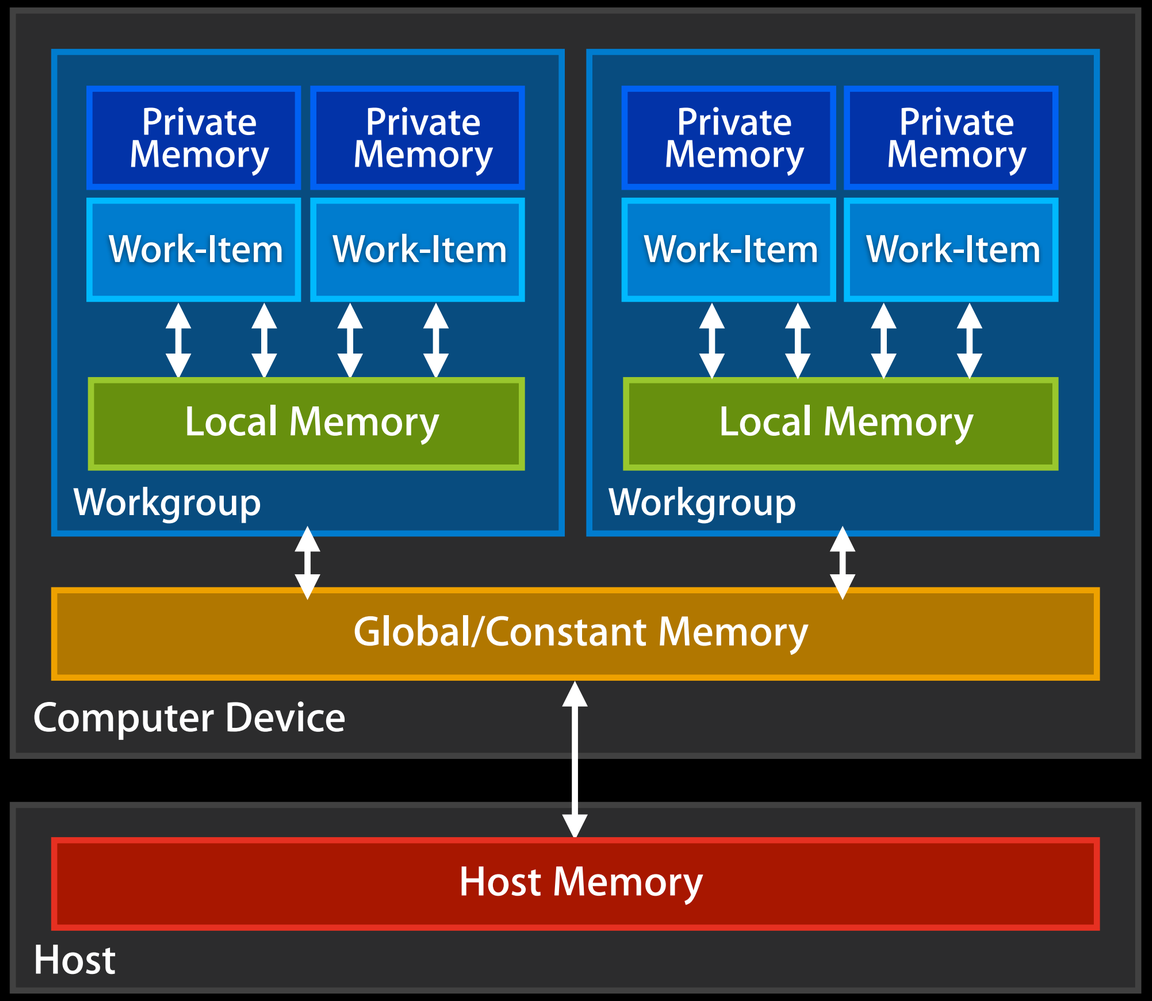
Can **not** synchronize outside of a *workgroup*

Problem Dimensions

- 1D: 1 million elements in an array:
  `global_dim[3] = {1000000,1,1};`
- 2D: 1920 x 1200 HD video frame, 2.3M pixels:
  `global_dim[3] = {1920, 1200, 1};`
- 3D: 256 x 256 x 256 volume, 16.7M voxels:
  `global_dim[3] = {256, 256, 256};`

- Choose the dimensions that are "best" for your algorithm
  - Maps well
  - Performs well

OpenCL Memory Model



- Private Memory: Per *work-item*
- Local Memory: Shared within a *workgroup* (16Kb)
- Local Global/Constant Memory: Not synchronized (128–512MB)
- Host Memory: On the CPU

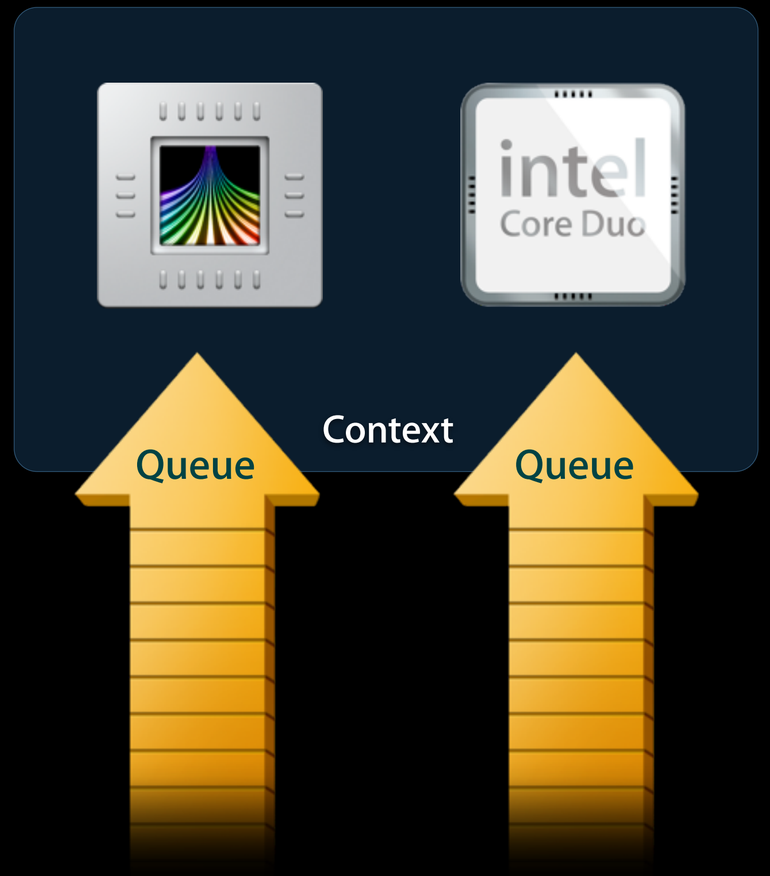
Memory management is explicit

You **must move data from host → global → local** ***and back***

Setup and Allocation of Resources

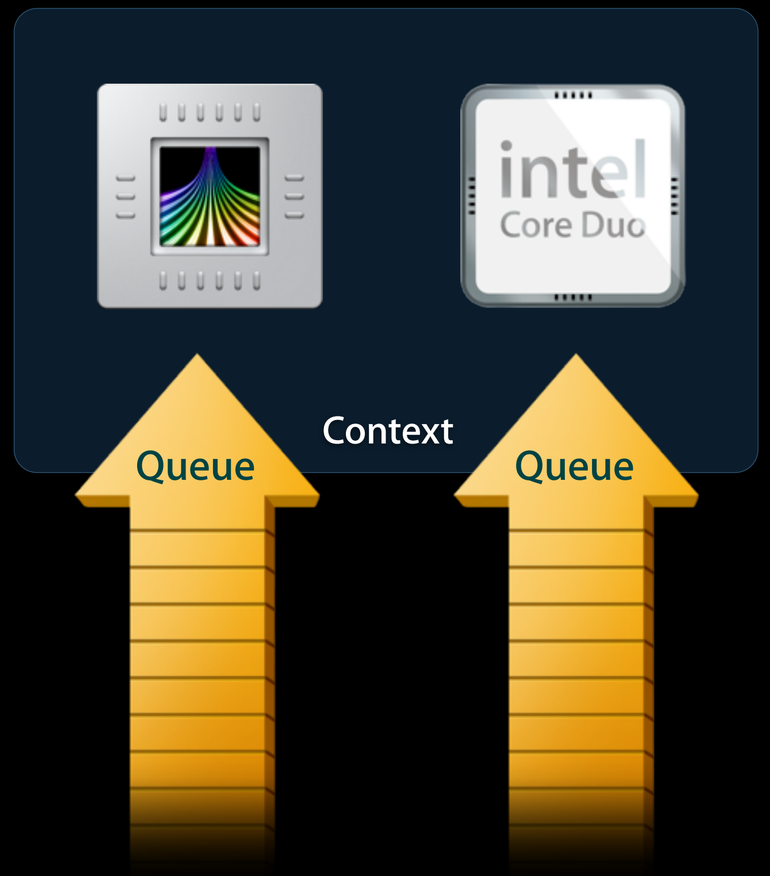
OpenCL Objects

- Setup
  - Devices—GPU, CPU
  - Contexts—Collection of devices
  - Queues—Submit work to the device
- Memory
  - Buffers—Blocks of memory
  - Images—2D or 3D formatted images
- Execution
  - Programs—Collections of kernels
  - Kernels—Argument/execution instances
- Synchronization/profiling
  - Events



Setup

1. Get the device(s)
2. Create a context
3. Create command queue(s)




```
cl_uint num_devices_returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
                     &devices[0], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1,
                     &devices[1], &num_devices_returned);
cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Setup: Notes

- Devices
  - All CPU cores together are a single device
  - OpenCL executes kernels across all cores in a data-parallel manner
- Contexts
  - Enable sharing of memory between devices
  - To share between the CPU and GPU, both must be in the same context
- Queues
  - All work submitted through queues
  - Each device must have a queue

Choosing Devices

- A system may have several devices—which is best?
- The “best” device is algorithm- and hardware-dependent

- Query device info with: `clGetDeviceInfo`(*device, param_name, *value*)
  - Number of compute units `CL_DEVICE_MAX_COMPUTE_UNITS`
  - Clock frequency `CL_DEVICE_MAX_CLOCK_FREQUENCY`
  - Memory size `CL_DEVICE_GLOBAL_MEM_SIZE`
  - Extensions (double precision, atomics, etc.)

- Pick the best device for your algorithm

Memory Resources

- Buffers
  - Simple chunks of memory
  - Kernels can access however they like (array, pointers, structs)
  - Kernels can read and write buffers
- Images
  - Opaque 2D or 3D formatted data structures
  - Kernels access only via read_image() and write_image()
  - Each image can be read or written in a kernel, but not both

Image Formats and Samplers

- Formats
  - Channel orders: `CL_A, CL_RG, CL_RGB, CL_RGBA`, etc.
  - Channel data type: `CL_UNORM_INT8, CL_FLOAT`, etc.
  - `clGetSupportedImageFormats()` returns supported formats

- Samplers (for reading images)
  - Filter mode: Linear or nearest
  - Addressing: Clamp, clamp-to-edge, repeat, or none
  - Normalized: True or false

- Benefit from image access hardware on GPUs

Allocating Images and Buffers

```
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA;

cl_mem input_image;
input_image  = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
                               image_width, image_height, 0, NULL, &err);
cl_mem output_image;
output_image = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
                               image_width, image_height, 0, NULL, &err);

cl_mem input_buffer;
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              sizeof(cl_float)*4*image_width*image_height, NULL, &err);
cl_mem output_buffer;
output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                               sizeof(cl_float)*4*image_width*image_height, NULL, &err);
```

Reading and Writing Memory Objects

- Explicit commands to access memory objects
  - `clEnqueueReadImage(object, blocking, origin, region, *ptr, ...)`
  - `clEnqueueWriteImage(object, blocking, origin, region, *ptr, ...)`
- Operate synchronously (`blocking = CL_TRUE`) or asynchronously

```
cl_float *input_data = (cl_float*)malloc(sizeof(float)*4*image_width*image_height);

size_t origin[3] = {0,0,0};
size_t region[3] = {image_width, image_height, 1};

err = clEnqueueReadImage(queue, input_image, CL_TRUE, origin, region, 0, 0,
                         input_data, 0, NULL, NULL);

err = clEnqueueReadBuffer(queue, input_buffer, CL_TRUE, 0,
          sizeof(cl_float)*4*image_width*image_height, input_data, 0, NULL, NULL);
```

Compilation and Execution of Kernels

Compiling Kernels

1. Create a program
   - Input: String (source code) or precompiled binary
   - Analogous to a dynamic library: A collection of kernels
2. Compile the program
   - Specify the devices for which kernels should be compiled
   - Pass in compiler flags
   - Check for compilation/build errors
3. Create the kernels
   - Returns a kernel object used to hold arguments for a given execution

Creating a Program

File: kernels.cl

```
// ---------------------------------
// Images Kernel
// ---------------------------------
__kernel average_images(read_only image2d_t input,
                write_only image2d_t output)
{
  sampler_t sampler = CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;
  int x = get_global_id(0);
  int y = get_global_id(1);
  float4 sum = (float4)0.0f;

  int2 pixel;
  for (pixel.x=x-SIZE; pixel.x<=x+SIZE; pixel.x++)
    for (pixel.y=y-SIZE; pixel.y<=y+SIZE; pixel.y++)
    sum += read_imagef(input, sampler, pixel);

  write_imagef(output, (int2)(x, y), sum/TOTAL);
};

...
```

```
cl_program program;
program = clCreateProgramWithSource(context, 1, &source, NULL, &err);
```

Compiling and Creating a Kernel

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

if (err) {
  char log[10240] = "";
  err = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                              sizeof(log), log, NULL);
  printf("Program build log:\n%s\n", log);
}
kernel = clCreateKernel(program, "average_images", &err);
```

Executing Kernels

1. Set the kernel arguments
2. Enqueue the kernel

```
err = clSetKernelArg(kernel, 0, sizeof(input), &input);
err = clSetKernelArg(kernel, 1, sizeof(output), &output);
```

```
size_t global[3] = {image_width, image_height, 0};
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
```

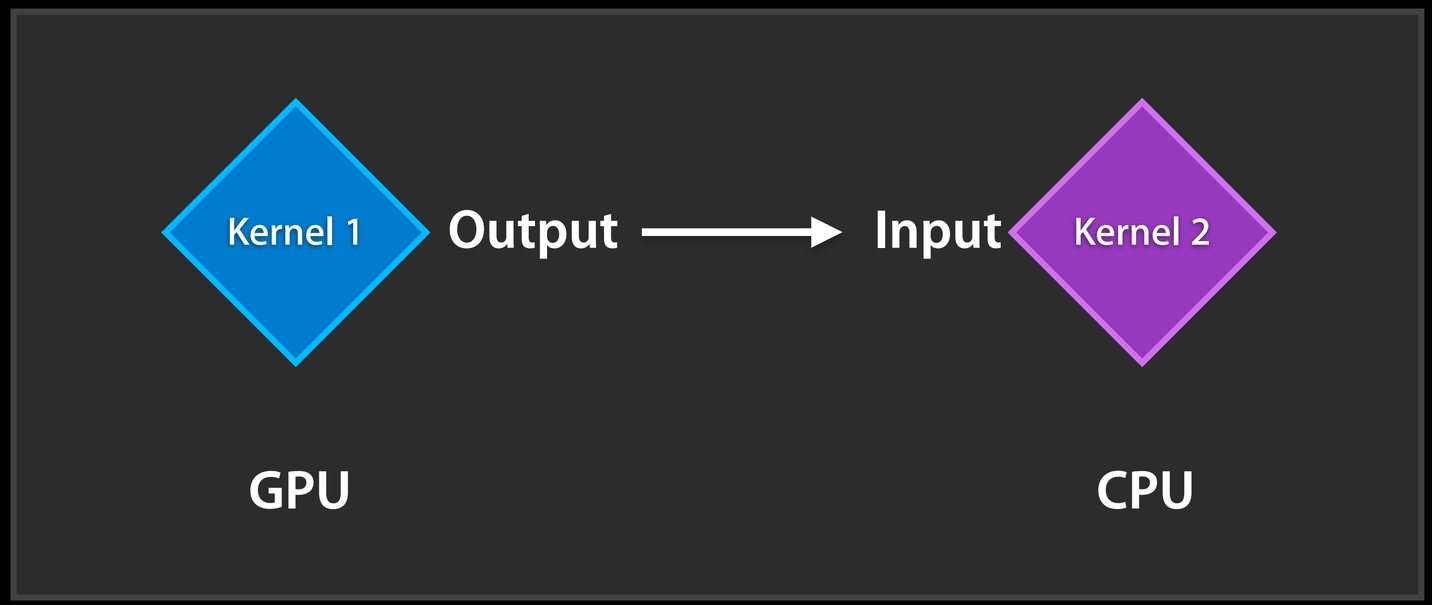
- Note: Your kernel is executed ***asynchronously***
  - Nothing may happen—you have just enqueued your kernel
  - Use a blocking read `clEnqueueRead*(... CL_TRUE ...)`
  - Use events to track the execution status

Synchronization Between Commands

- Each *individual* queue is in order
  - All items execute in order
  - Behaves as expected (as long as you're enqueuing from one thread)
- You must *explicitly synchronize between queues*
  - Multiple devices each have their own queue
  - Use events to synchronize

- Events
  - Commands ***return events*** and ***obey waitlists***
  - `clEnqueue*(..., num_events_in_waitlist, *event_waitlist, *event_out)`

Synchronization: Two Devices/Queues

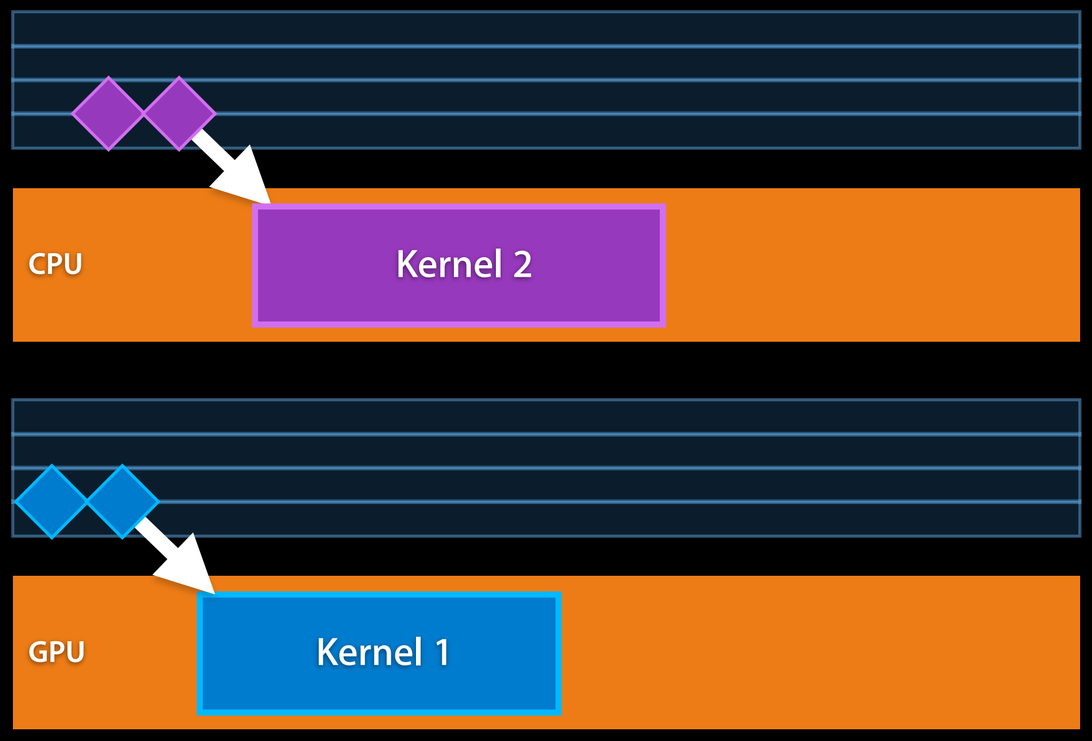


Explicit dependency: Kernel 1 must *finish* before Kernel 2 *starts*

Synchronization: Two Devices/Queues

Enqueue Kernel 1
Enqueue Kernel 2

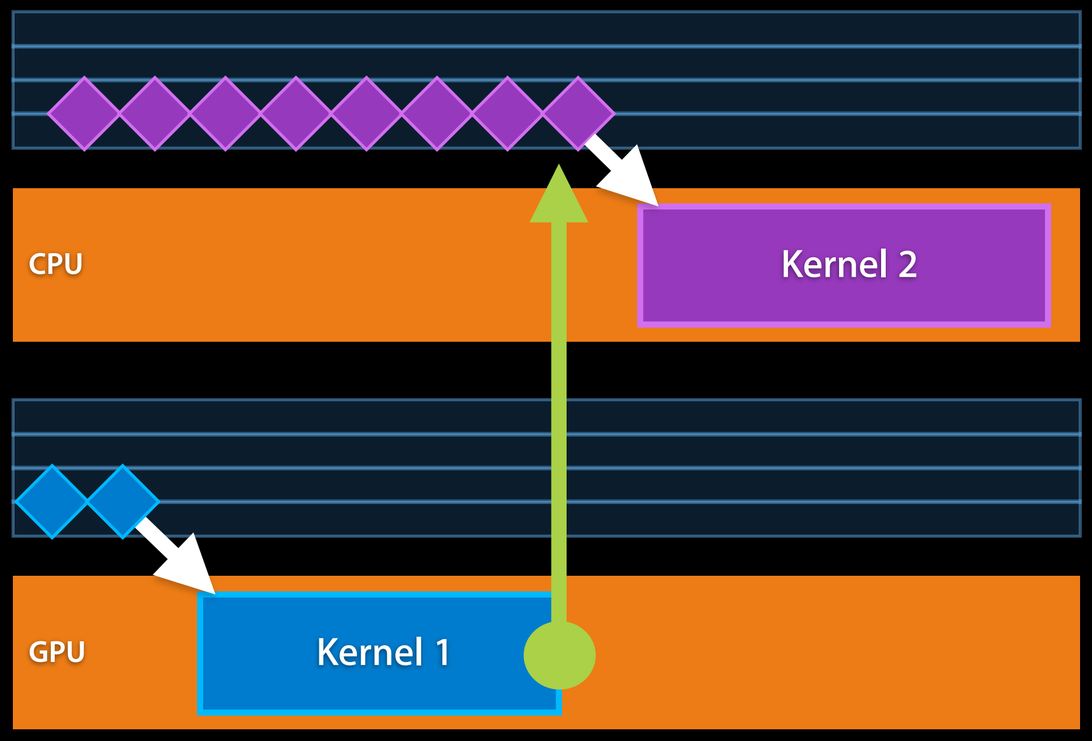
Kernel 2 starts before the results from Kernel 1 are ready



Time ⟶

Enqueue Kernel 1
Enqueue Kernel 2

Kernel 2 waits for an event from Kernel 1, and does not start until the results are ready



Time ⟶

Using Events on the Host

- `clWaitForEvents(num_events, *event_list)`
  - Blocks until events are complete
- `clEnqueueMarker(queue, *event)`
  - Returns an event for a marker that moves through the queue
- `clEnqueueWaitForEvents(queue, num_events, *event_list)`
  - Inserts a “WaitForEvents” into the queue
- `clGetEventInfo()`
  - Command type and status
    `CL_QUEUED, CL_SUBMITTED, CL_RUNNING, CL_COMPLETE`, or error code
- `clGetEventProfilingInfo()`
  - Command queue, submit, start, and end times

Demo

Hints

Performance and Debugging

Performance: Overhead

- Compiling programs is very expensive
  - Reuse programs or precompile binaries
- Moving data to/from the GPU is very expensive
  - Keep data on the GPU
- Starting a kernel is expensive
  - Do a lot of work for each execution
- Events on the GPU are expensive
  - Only use events where needed

Performance: Kernels and Memory

- Large global work sizes help hide memory latency
  - 1000+ work-items preferred
- Trade-off math precision and performance with half_ and native_
- Divergent execution can be bad on GPUs
  - All work-items should take very similar control flow paths
  - Not a problem on CPUs
- Handle data reuse through local memory when available
- Access memory sequentially across work-items
  - Enables hardware memory coalescing
  - Dramatic bandwidth improvements

Debugging

- Start on the CPU
  - You can use printf()
  - You can use shark
  - You can look at assembly in gdb
- Be very careful about reading/writing out of bounds on the GPU
  - Use explicit address checks around reads and writes if a kernel is crashing to locate problems
- Play nicely with other apps
  - GPUs are not preemptively scheduled
- Use extra output buffers/images to record intermediate values
- Set a context call-back function to report API errors
  - Set `CL_LOG_ERRORS=stdout` or `stderr` or `syslog`
  - Or use your own

OpenCL Kernel Language

Mon Ping Wang

OpenCL C Language

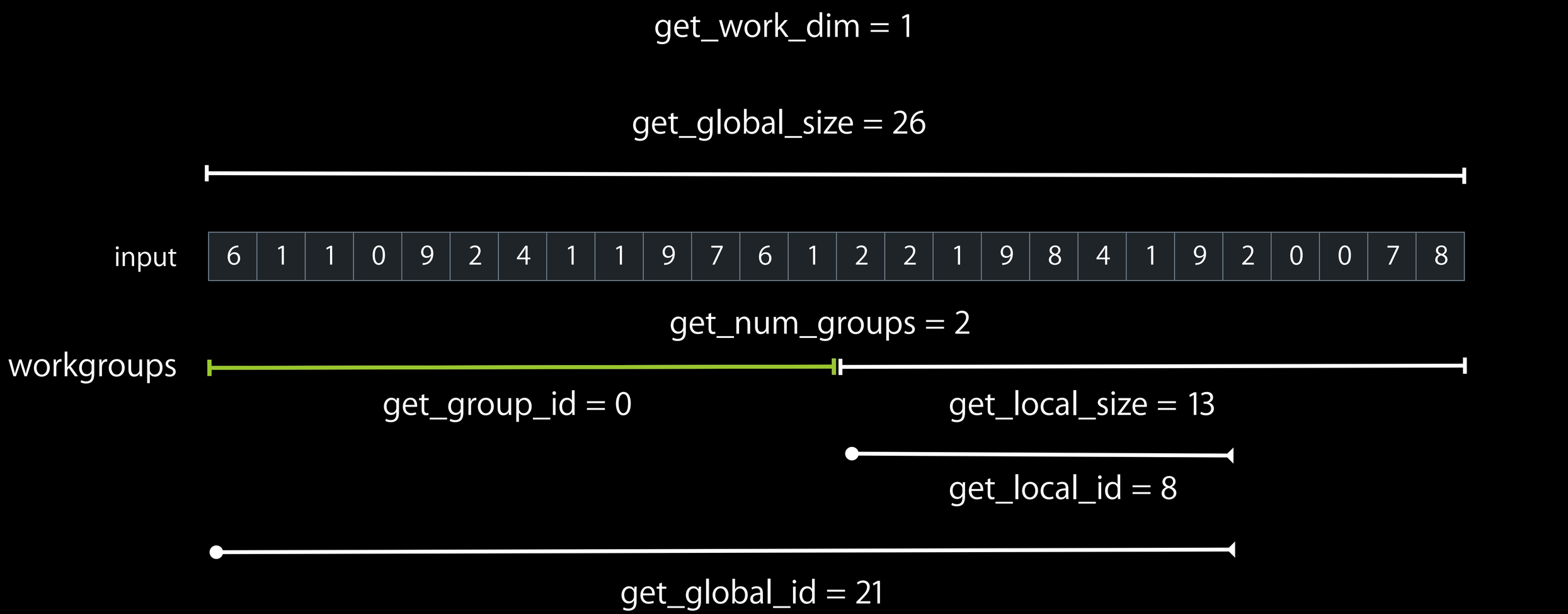
- Derived from ISO C99
  - No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- Parallel extensions
  - Work-items and workgroups
  - Vector types
  - Synchronization
- Address spaces
- Optimized image access
- Built-in functions

Kernel

- A data-parallel function executed for each work-item

```
kernel void square(global float* input, global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```

Work-Items and Workgroup Functions



Data Types

- Scalar data types
  - char , uchar, short, ushort, int, uint, long, ulong
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types

Vector Types

- Portable
- Vector length of 2, 4, 8, and 16
- char2, ushort4, int8, float16, double2, …
- Endian safe
- Aligned at vector length
- Vector operations and built-in functions

Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

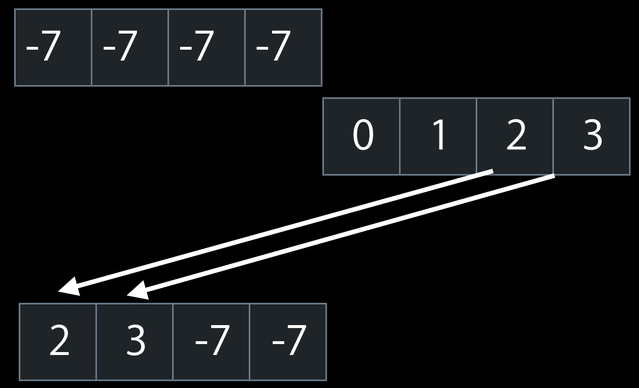
Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

- Vector components

```
vi0.lo = vi1.hi;
```



Vector Operations

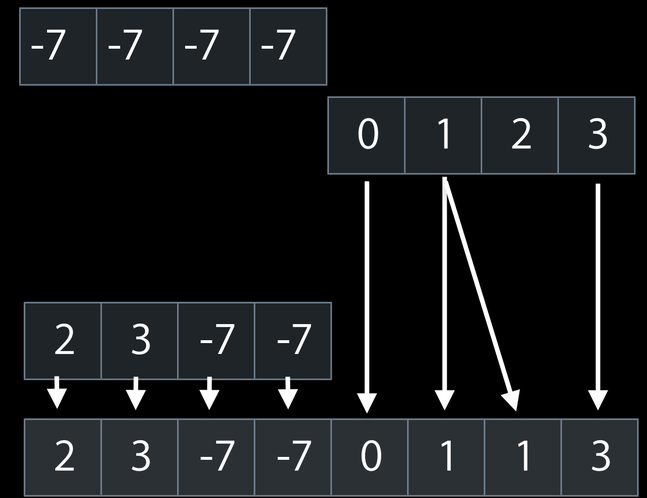
- Vector literal

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

- Vector components

```
vi0.lo = vi1.hi;
```

```
int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```



Vector Operations

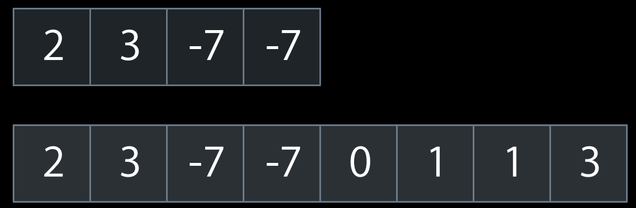
- Vector literal

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

- Vector components

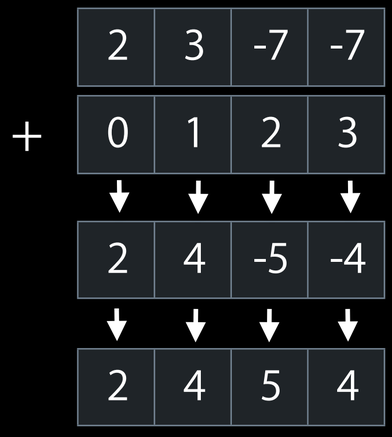
```
vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```



- Vector ops

```
vi0 += vi1;
```




```
vi0 = abs(vi0);
```

Workgroup Functions

- Workgroup functions
  - Encountered by all work-items in the workgroup
  - With the same argument values
- Synchronization
  - Barrier
    - mem_fence, read_mem_fence, write_mem_fence
- async_work_group_copy

```
kernel read(global int* g, local int* shared) {
  if (get_global_id(0) < 5)
    mem_fence(CLK_GLOBAL_MEM_FENCE);
  else
    k = array[0];
}
```

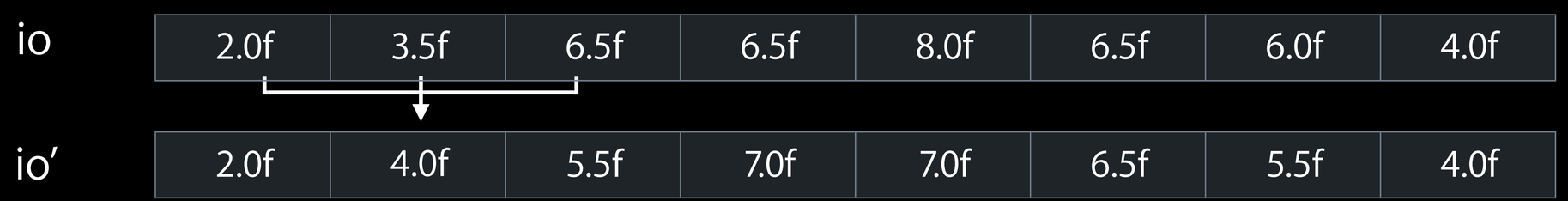
work-item 0

work-item 6

Illegal since not all work-items encounter mem_fence

Barrier Example

```
kernel void smooth(global float* io) {
  float temp;
  int id = get_global_id(0);
  temp = (io[id-1] + id[id] + io[id+1])/3.0f;
  barrier(CLK_GLOBAL_MEM_FENCE);
  io[id] = temp;
}
```

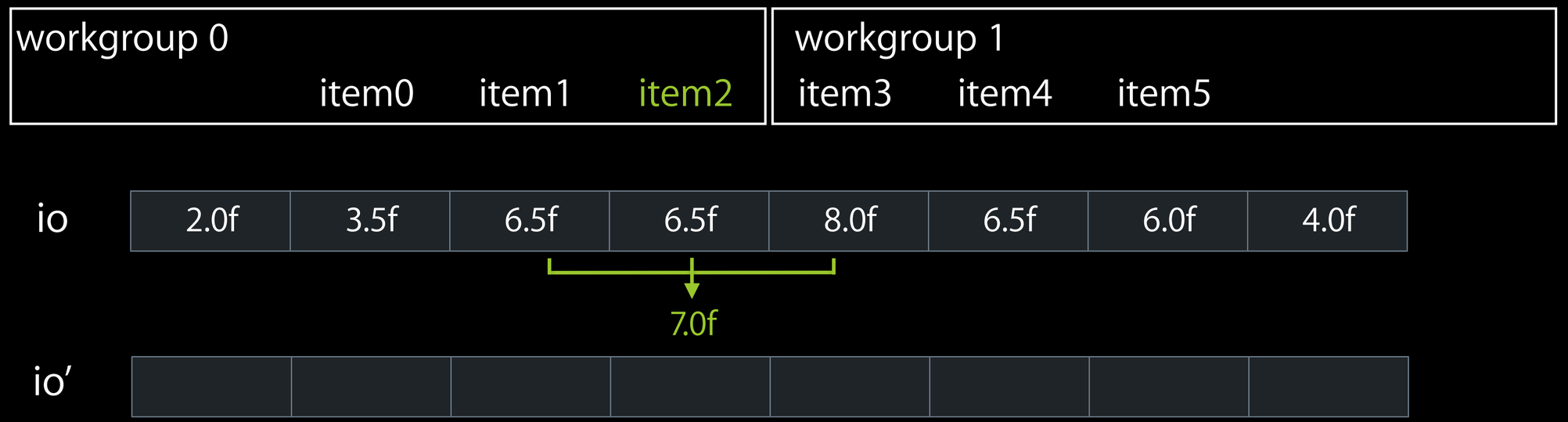


Barrier Example

```
kernel void smooth(global float* io) {
  float temp;
  int id = get_global_id(0);
  temp = (io[id-1] + id[id] + io[id+1])/3.0f;
  barrier(CLK_GLOBAL_MEM_FENCE);
  io[id] = temp;
}
```

item0, 1, 3, 4, 5

item2



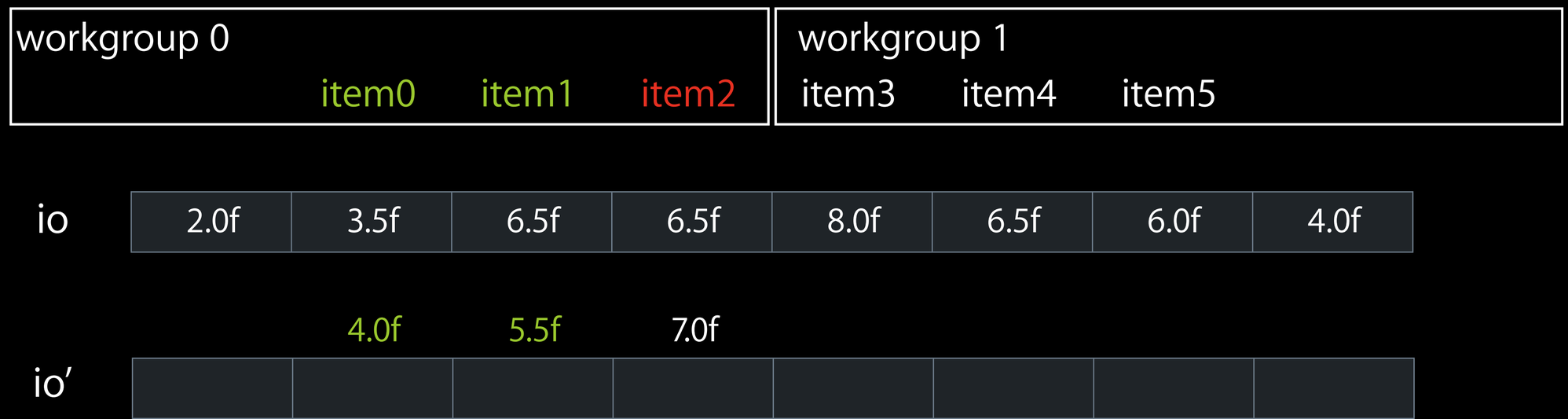
Barrier Example

```
kernel void smooth(global float* io) {
  float temp;
  int id = get_global_id(0);
  temp = (io[id-1] + id[id] + io[id+1])/3.0f;
  barrier(CLK_GLOBAL_MEM_FENCE);
  io[id] = temp;
}
```

item3, 4, 5

item0 & 1

item2

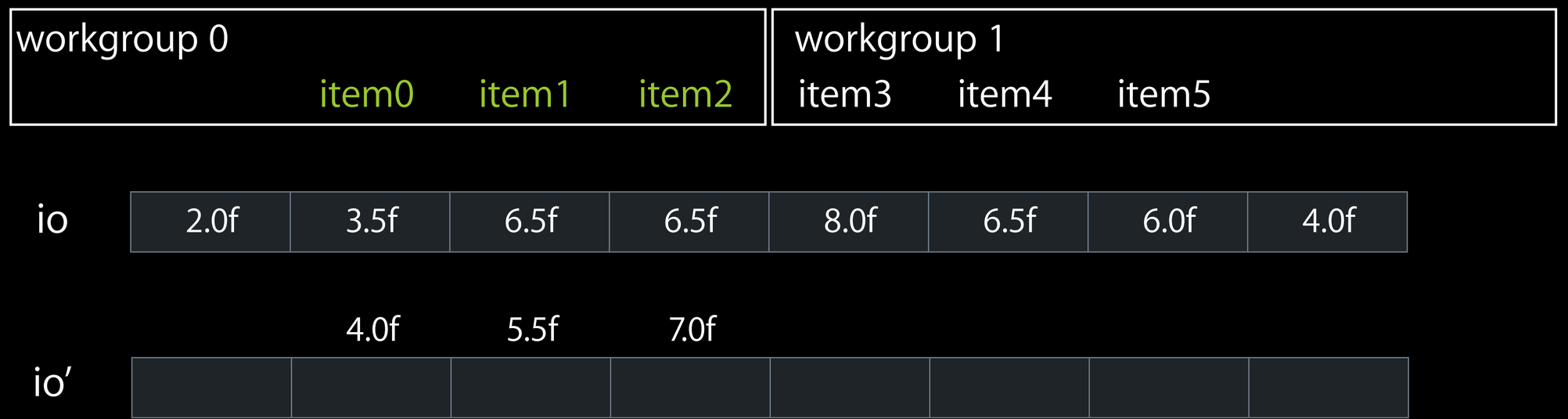


Barrier Example

```
kernel void smooth(global float* io) {
  float temp;
  int id = get_global_id(0);
  temp = (io[id-1] + id[id] + io[id+1])/3.0f;
  barrier(CLK_GLOBAL_MEM_FENCE);
  io[id] = temp;
}
```

item3, 4, 5

item0,1, 2



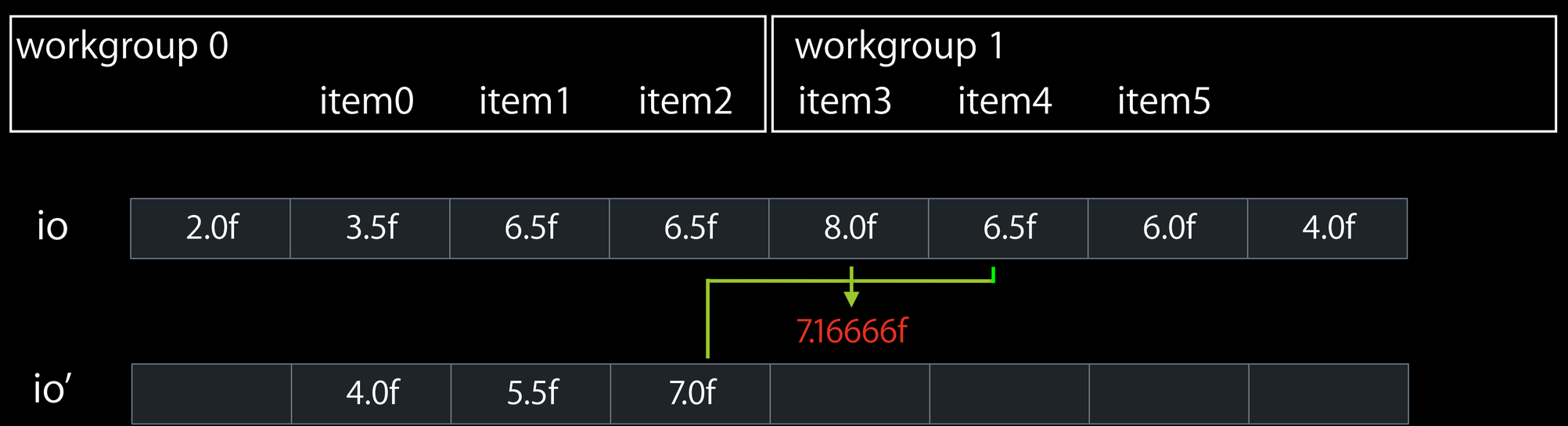
Barrier synchronizes only work-items within a workgroup

Barrier Example

```
kernel void smooth(global float* io) {
  float temp;
  int id = get_global_id(0);
  temp = (io[id-1] + id[id] + io[id+1])/3.0f;
  barrier(CLK_GLOBAL_MEM_FENCE);
  io[id] = temp;
}
```

item 4, 5

item3

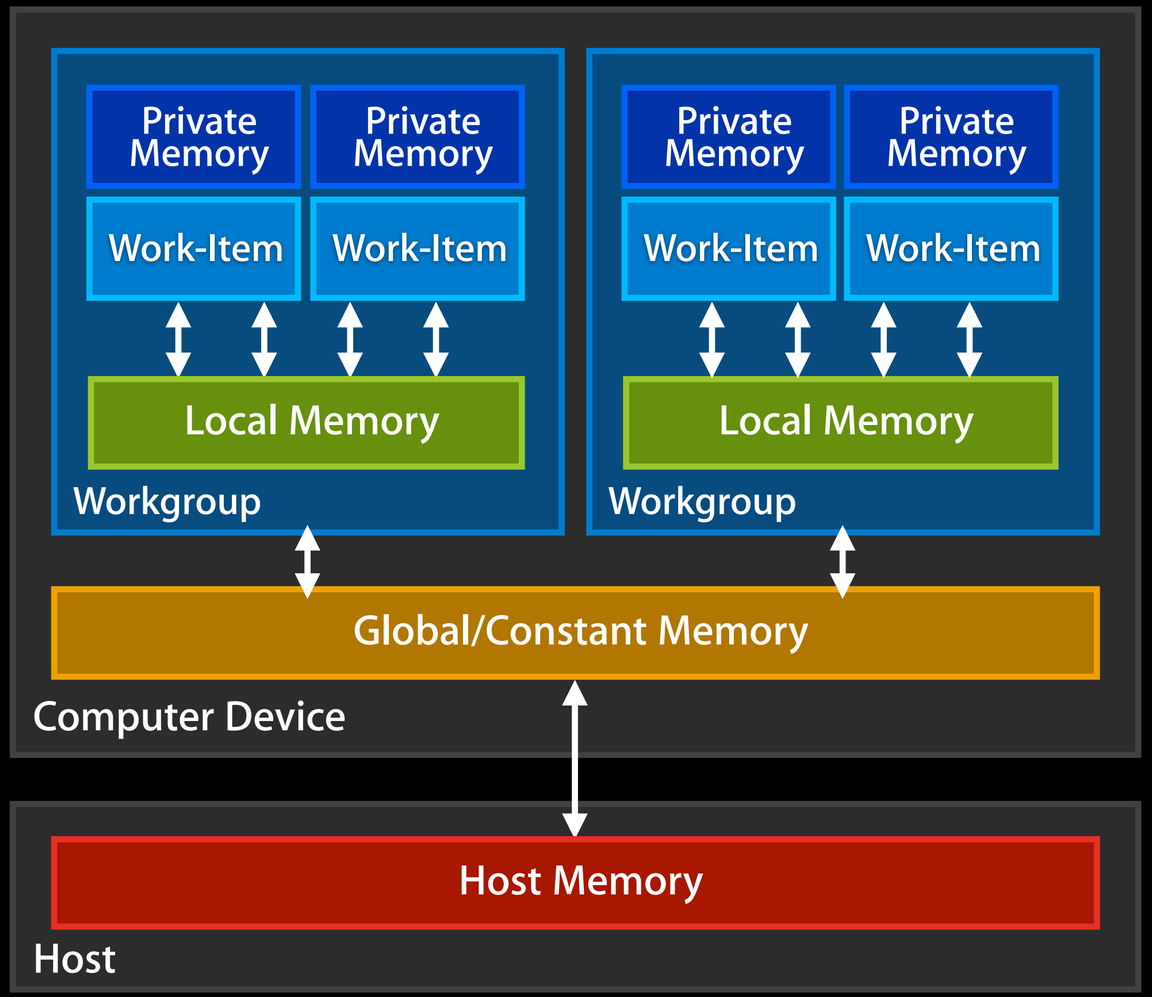


Barrier synchronizes only work-items within a workgroup

Synchronization

- Atomic functions to global and local memory
  - Add, sub, xchg, inc, dec, cmp_xchg, min, max, and, or, xor
  - 32-bit/64-bit integers
  - Extension: Check with `clGetDeviceInfo` with `CL_DEVICE_EXTENSIONS`

Address Spaces



- Private: Private memory per *work-item*
- Local: Shared local memory per *workgroup*
- Constant: Constant memory
- Global: Global memory

Address Spaces

- Kernel pointer arguments must use global, local, or constant

```
kernel void distance(global float8* stars, local float8* local_stars)
kernel void sum(private int* p)   // Illegal because is uses private
```

- Default address space for arguments and local variables is private

```
kernel void smooth(global float* io) {
  float temp;
  ...
```

- image2d_t and image3d_t are always in global address space

```
kernel void average(read_only global image_t in, write_only image2d_t out)
```

Address Spaces

- Program (global) variables must be in constant address space

```
constant float bigG = 6.67428E-11;
global   float time;              // Illegal non constant
kernel void force(global float4 mass) { time = 1.7643E18f; }
```

- Casting between different address spaces is undefined

```
kernel void calcEMF(global float4* particles) {
  global float* particle_ptr = (global float*) particles;
  float* private_ptr = (float*) particles; // Undefined behavior -
  float particle = * private_ptr;          //  different address
}
```

Average Kernel

```
// Average a 7x7 region in an image using buffers
kernel average(global float4 *input, global float4 *output,
               int width, int height) {
  int x = get_global_id(0);
  int y = get_global_id(1);
  float4 sum = (float4)0.0f;

  int2 pixel;
  for (pixel.x=x-3; pixel.x<=x+3; pixel.x++)
    for (pixel.y=y-3; pixel.y<=y+3; pixel.y++)
      if ((pixel.x>=0 && pixel.x<width) && (pixel.y>=0 && pixel.y<height))
        sum += input[pixel.y*width + pixel.x];
      output[y*width+x] = sum/49.0f;\n"
}
```

Local Memory

- Local memory may be much faster on certain devices

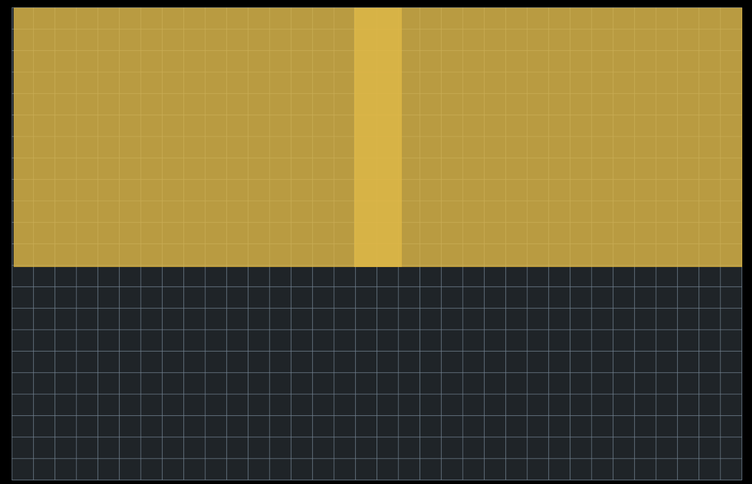
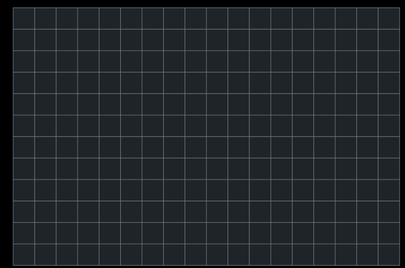


Image in Global Memory



Local Memory

- async_work_group_copy

Local Memory Example

```
 // Average a 7x7 region in an image
kernel average_local(global float4* input, global float4* output,
                     int width, int height, local float4* tiled)
{
  int total_image_width = width+6;
  total_tile_width = get_local_size(0)+6;
  int y_tile_loc = get_group_id(1)*get_local_size(1);
  int x_tile_loc = get_group_id(0)*get_local_size(0);
  for (int i=0; i < get_local_size(1)+6; i++) {
    event_t t = async_work_group_copy(&tile[total_tile_width*i],
                  &input[total_image_width*(y_tile_loc+i) + x_tile_loc],
                  total_tile_width, 0);
    wait_group_events(1, &t);
  }
  output[get_global_id(1)*width + get_global_id(0)] = ComputeAverage(tiled);
}
```

(Setup address for tiles)

Images

- Image functions [float, integer, unsigned integer]
  - read_image[f | i | ui]
  - write_image[f | i | ui]
  - get_image[width | height | depth]
  - get_image_channel_data_type
  - get_image_channel_order

Conversions

- Scalar and pointer conversions follow C99 rules
- No implicit conversions

```
float4 f4 = int4_vec;              // Illegal implicit conversion
```

- No casts for vector types (different semantics for vectors)

```
float4 f4 = (float4) int4_vec;  // Illegal cast
```

- Casts has other problems

```
float x;
int i = (int)(x + 0.5f);          // Round float to nearest integer
```

Wrong for:
0.5f - 1 ulp (rounds up not down)
negative numbers (wrong answer)

- There is hardware to do it on nearly every machine

Conversions

- Explict conversions: convert_*destType*<_sat><_roundingMode>]
  - Scalar and vector types
  - No ambiguity

```
uchar4 c4 = convert_uchar4_sat_rte(f4);
```

f4	-5.0f	254.5f	254.6	1.2E9f
c4	0	254	255	255

Saturate to 0

Round down to nearest even

Round up to nearest value

Saturated to 255

Reinterpret Data: as_*typen*

- Reinterpret the bits to another type
- Types must be the same size

```
// f[i] = f[i] < g[i] ? f[i] : 0.0f
float4 f, g;
int4 is_less = f < g;
f = as_float4(as_int4(f) & is_less);
```

f	-5.0f	254.5f	254.6f	1.2E9f
g	254.6f	254.6f	254.6f	254.6f
is_less	ffffffff	ffffffff	00000000	00000000
as_int	c0a00000	42fe0000	437e8000	4e8f0d18
&	c0a00000	42fe0000	00000000	00000000
f	-5.0f	254.5f	0.0f	0.0f

OpenCL provides a Select built-in

Built-in Functions

Math Functions
gentype acos (gentype)
gentype acosh (gentype)
gentype acospi (gentype x)
gentype asin (gentype)
gentype asinh (gentype)
gentype asinpi (gentype x)
gentype atan (gentype y_over_x)
gentype atan2 (gentype y, gentype x)
gentype atanh (gentype)
gentype atanpi (gentype x)
gentype atan2pi (gentype y, gentype x)
gentype cbrt (gentype)
gentype ceil (gentype)
gentype copysign (gentype x, gentype y)
gentype cos (gentype)
gentype cosh (gentype)
gentype cospi (gentype x)
gentype erfc (gentype)
gentype erf (gentype)
gentype exp (gentype x)
gentype exp2 (gentype)
gentype exp10 (gentype)
gentype expm1 (gentype x)
gentype fabs (gentype)
gentype fdim (gentype x, gentype y)
gentype floor (gentype)
gentype fma (gentype a, gentype b, gentype c)
gentype fmax (gentype x, gentype y)
gentype fmax (gentype x, float y)
gentype fmin (gentype x, gentype y)
gentype fmin (gentype x, float y)
gentype fmod (gentype x, gentype y)
gentype fract (gentype x, gentype *iptr)27
gentype frexp (gentype x, intn *exp)
gentype hypot (gentype x, gentype y)
intn ilogb (gentype x)
gentype ldexp (gentype x, intn n)
gentype ldexp (gentype x, int n)
gentype lgamma (gentype x)
gentype lgamma_r (gentype x,
gentype log (gentype)
gentype log2 (gentype)
gentype log10 (gentype)
gentype log1p (gentype x)
gentype logb (gentype x)
gentype mad (gentype a, gentype b, gentype c)
gentype modf (gentype x, gentype *iptr)
gentype nan (uintn nancode)
gentype nextafter (gentype x, gentype y)
gentype pow (gentype x, gentype y)
gentype pown (gentype x, intn y)
gentype powr (gentype x, gentype y)
gentype remainder (gentype x, gentype y)
gentype remquo (gentype x, gentype y, intn *quo)
gentype rint (gentype)
gentype rootn (gentype x, intn y)
gentype round (gentype x)
gentype rsqrt (gentype)
gentype sin (gentype)
gentype sincos (gentype x, gentype *cosval)
gentype sinh (gentype)
gentype sinpi (gentype x)
gentype sqrt (gentype)
gentype tan (gentype)
gentype tanh (gentype)
gentype tanpi (gentype x)
gentype tgamma (gentype)
gentype trunc (gentype)
Integer Ops
ugentype abs (gentype x)
ugentype abs_diff (gentype x, gentype y)
gentype add_sat (gentype x, gentype y)
gentype hadd (gentype x, gentype y)
gentype rhadd (gentype x, gentype y)29
gentype clz (gentype x)
gentype mad_hi (gentype a, gentype b, gentype c)
gentype mad_sat (gentype a, gentype b, gentype c)
gentype max (gentype x, gentype y)
gentype min (gentype x, gentype y)
gentype mul_hi (gentype x, gentype y)
gentype rotate (gentype v, gentype i)
gentype sub_sat (gentype x, gentype y)
shortn upsample (intn hi, uintn lo)
ushortn upsample (uintn hi, uintn lo)
intn upsample (intn hi, uintn lo)
uintn upsample (uintn hi, uintn lo)
longn upsample (intn hi, uintn lo)
ulongnn upsample (uintn hi, uintn lo)
gentype mad24 (gentype x, gentype y, gentype z)
gentype mul24 (gentype x, gentype y)
Common Functions
gentype clamp (gentype x, gentype minval, gentype maxval)
gentype clamp (gentype x, float minval, float maxval)
gentype degrees (gentype radians)
gentype max (gentype x, gentype y)
gentype max (gentype x, float y)
gentype min (gentype x, gentype y)
gentype min (gentype x, float y)
gentype mix (gentype x, gentype y, gentype a)
gentype mix (gentype x, gentype y, float a)
gentype radians (gentype degrees)
gentype sign (gentype x)
Geometric Functions
float4 cross (float4 p0, float4 p1)
float dot (gentype p0, gentype p1)
float distance (gentype p0, gentype p1)
float length (gentype p)
float fast_distance (gentype p0, gentype p1)
float fast_length (gentype p)
gentype fast_normalize (gentype p)
Relational Ops
int isequal (float x, float y)
intn isequal (floatn x, floatn y)
int isnotequal (float x, float y)
intn isnotequal (floatn x, floatn y)
int isgreater (float x, float y)
intn isgreater (floatn x, floatn y)
int isgreaterequal (float x, float y)
intn isgreaterequal (floatn x, floatn y)
int isless (float x, float y)
intn isless (floatn x, floatn y)
int islessequal (float x, float y)
intn islessequal (floatn x, floatn y)
int islessgreater (float x, float y)
intn islessgreater (floatn x, floatn y)
int isfinite (float)
intn isfinite (floatn)
int isnan (float)
intn isnan (floatn)
int isnormal (float)
intn isnormal (floatn)
int isordered (float x, float y)
intn isordered (floatn x, floatn y)
int isunordered (float x, float y)
intn isunordered (floatn x, floatn y)
int signbit (float)
intn signbit (floatn)
int any (igentype x)
int all (igentype x)
gentype bitselect (gentype a, gentype b, gentype c)
gentype select (gentype a, gentype b,igentype c)
gentype select (gentype a, gentype b,ugentype c)
Vector Loads/Store Functions
gentypen vloadn (size_t offset, const global gentype *p)
gentypen vloadn (size_t offset, const __local gentype *p)
gentypen vloadn (size_t offset, const __constant gentype *p)
gentypen vloadn (size_t offset, const __private gentype *p)
void vstoren (gentypen data, size_t offset, global gentype *p)
void vstoren (gentypen data, size_t offset, __local gentype *p)
void vstoren (gentypen data, size_t offset, __private gentype *p)
void vstore_half (float data, size_t offset, global half *p)
void vstore_half_rte (float data, size_t offset, global half *p)
void vstore_half_rtz (float data, size_t offset, global half *p)
void vstore_half_rtp (float data, size_t offset, global half *p)
void vstore_half_rtn (float data, size_t offset, global half *p)
void vstore_half (float data, size_t offset, __local half *p)
void vstore_half_rte (float data, size_t offset, __local half *p)
void vstore_half_rtz (float data, size_t offset, __local half *p)
void vstore_half_rtp (float data, size_t offset, __local half *p)
void vstore_half_rtn (float data, size_t offset, __local half *p)
void vstore_half (float data, size_t offset, __private half *p)
void vstore_half_rte (float data, size_t offset, __private half *p)
void vstore_half_rtz (float data, size_t offset, __private half *p)
void vstore_half_rtp (float data, size_t offset, __private half *p)
void vstore_half_rtn (float data, size_t offset, __private half *p)
void vstore_halfn (floatn data, size_t offset, global half *p)
void vstore_halfn_rte (floatn data, size_t offset, global half *p)
void vstore_halfn_rtz (floatn data, size_t offset, global half *p)
void vstore_halfn_rtp (floatn data, size_t offset, global half *p)
void vstore_halfn_rtn (floatn data, size_t offset, global half *p)
void vstore_halfn (floatn data, size_t offset, __local half *p)
void vstore_halfn_rte (floatn data, size_t offset, __local half *p)
void vstore_halfn_rtz (floatn data, size_t offset, __local half *p)
void vstore_halfn_rtp (floatn data, size_t offset, __local half *p)
void vstore_halfn_rtn (floatn data, size_t offset, __local half *p)
void vstore_halfn (floatn data, size_t offset, __private half *p)
void vstore_halfn_rte (floatn data, size_t offset, __private half *p)
void vstore_halfn_rtz (floatn data, size_t offset, __private half *p)
void vstore_halfn_rtp (floatn data, size_t offset, __private half *p)
void vstore_halfn_rtn (floatn data, size_t offset, __private half *p)
void vstorea_halfn (floatn data, size_t offset, global half *p)
void vstorea_halfn_rte (floatn data, size_t offset, global half *p)
void vstorea_halfn_rtz (floatn data, size_t offset, global half *p)
void vstorea_halfn_rtp (floatn data, size_t offset, global half *p)
void vstorea_halfn_rtn (floatn data, size_t offset, global half *p)
void vstorea_halfn (floatn data, size_t offset, __local half *p)
void vstorea_halfn_rte (floatn data, size_t offset, __local half *p)
void vstorea_halfn_rtz (floatn data, size_t offset, __local half *p)
void vstorea_halfn_rtp (floatn data, size_t offset, __local half *p)
void vstorea_halfn_rtn (floatn data, size_t offset, __local half *p)
void vstorea_halfn (floatn data, size_t offset, __private half *p)
void vstorea_halfn_rte (floatn data, size_t offset, __private half *p)
void vstorea_halfn_rtz (floatn data, size_t offset, __private half *p)
void vstorea_halfn_rtp (floatn data, size_t offset, __private half *p)
void vstorea_halfn_rtn (floatn data, size_t offset, __private half *p)

Built-in Functions

- Rich set
- IEEE 754 compatible rounding modes
- Defines maximum error of math functions
- Support both scalar and vector types
- Efficient—tuned for the hardware they are executed on

Math Functions

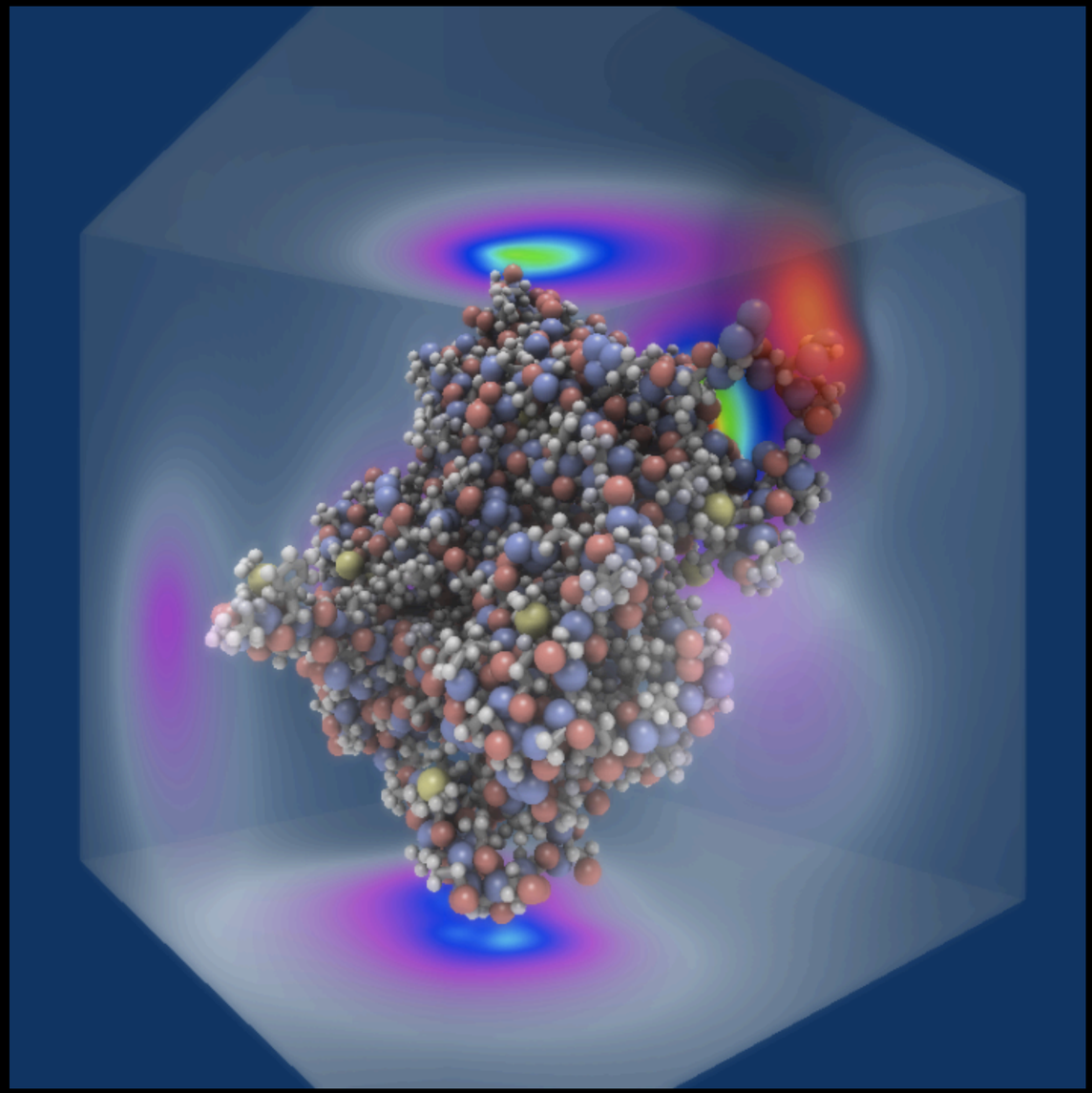
- Well defined—handles ambiguous C99 library edge cases
- 3 Flavors
  - Full precision
    - log <= 3ulp
  - Half precision—minimum 11 bits of accuracy/faster
    - half_log <= 8192 ulps
  - Native precision/fastest
    - native_log: implementation defined

Other Features

- Preprocessing directives defined by C99
- Optional extensions
  - Double precision floating point
  - Select rounding modes
  - Atomic functions
  - Writing 3D image objects
  - Etc.

Summary

- Portable high-performance framework
  - Ideal for data-parallel, computationally intensive algorithms
  - Access to all compute resources
  - Portable across different devices
- Well-defined computation/memory model
  - Global and local problem dimensions for describing parallelism
  - Multiple addresses spaces
- Efficient parallel programming language
  - C99 with extensions for task and data parallelism
  - Rich set of built-in functions



More Information

Allan Schaffer
2D and 3D Graphics Technologies Evangelist
aschaffer@apple.com

Related Sessions

Harnessing the Power of OpenCL	Marina Thursday 10:30AM
Mastering OpenCL	Marina Thursday 2:00PM

Labs

OpenCL Lab	Graphics and Media Lab B Thursday 3:30PM

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09